

## AN INTRODUCTION TO SOFTWARE ENGINEERING AND FAULT TOLERANCE

PATRIZIO PELLICCIONE and HENRY MUCCINI

*Dipartimento di Informatica, University of L'Aquila*

*Via Vetoio, 1, 67100 L'Aquila, ITALY*

*E-mail: pellicci, muccini@di.univaq.it*

NICOLAS GUELFI

*Laboratory for Advanced Software Systems, University of Luxembourg*

*6, rue Richard Coudenhove-Kalergi, LUXEMBOURG*

*E-mail: nicolas.guelfi@uni.lu*

ALEXANDER ROMANOVSKY

*School of Computing Science, Newcastle University*

*Newcastle upon Tyne, NE1 7RU, UK*

*E-mail: alexander.romanovsky@newcastle.ac.uk*

### 1. Motivations for the Book

Building systems that are trustful is one of the main challenges which software developers are facing. Dependability-related concerns have accompanied system developers since the first day these systems were built and deployed. Obviously various things have changed since then, including, the nature of faults and failures, the complexity of the systems, the services they deliver and the way our society uses these systems. But the need to deal with various threads (such as failed components, deteriorating environments, component mismatches, human mistakes, intrusions and software bugs) is still in the core of software and system research and development. As computers are now penetrating various new domains (including the critical ones) and the complexity of the modern systems is growing, achieving dependability remains central for system developers and users.

Accepting that errors always happen in spite of all the efforts to eliminate the faults that might cause them is in the core of dependability. To

to this end various fault tolerance mechanisms have been investigated by researchers and used in industry. Unfortunately, more often than not these solutions exclusively focus on the implementation (e.g. they are provided as middleware/OS services or libraries) ignoring other development phases, most importantly the earlier ones. This creates a dangerous gap between the requirements to build dependable (and fault tolerant) systems and trying to meet them by exclusively using specific fault tolerance mechanisms at the implementation step.<sup>1</sup> One of the consequences of this is that there is a growing number of reported situations in which fault tolerance means undermine the overall system dependability as they are not used properly.

We believe that fault tolerance needs to be explicitly included into the traditional software engineering theories and practices, and it should become a part of all steps of software development. As the current software engineering practices tend to capture only normal behaviour, assuming that all faults can be removed during development, new software engineering methods and tools need to be developed to support explicit handling of abnormal situations. Moreover, every phase in the software development process needs to be enriched with the phase-specific fault tolerance means. Generally speaking, integrating fault tolerance into software engineering requires:

- integrating fault tolerance means into system models starting from the early development phases (i.e. requirement and architecture);
- making fault tolerance-related decisions for each appropriate model at each phase by explicit modelling of faults, fault tolerance means and dedicated redundant resources (with a specific focus on fault tolerant software architectures);
- ensuring correct transformations of the models used at various development phases with a specific focus on transformation of the fault tolerance means;
- supporting verification and validation of the fault tolerance means;
- developing dedicated tools for fault tolerance modelling;
- providing domain-specific application-level fault tolerance mechanisms and abstractions.

This book consists of the chapters describing novel approaches to integrating fault tolerance into software development process. They cover a wide range of topics focusing on fault tolerance during the different phases of the software development, software engineering techniques for verification and validation of fault tolerance means, and languages for supporting

fault tolerance specification and implementation. Accordingly, the book is structured into the following three parts:

- Part A: *Fault tolerance engineering: from requirements to code*;
- Part B: *Verification and validation of fault tolerant systems*;
- Part C: *Languages and Tools for engineering fault tolerant systems*.

The next section of this chapter briefly introduces the main dependability and fault tolerance concepts. Section 3 defines the software engineering realm. Sections 4, 5 and 6 introduce the three areas corresponding to the book parts and briefly outline the current state or research. The last section summarises the content of the book.

## 2. Dependability and Fault Tolerance

Dependability is usually defined as the system ability to deliver service that can be justifiably trusted.<sup>2</sup> Ensuring the required dependability level for complex computer-based systems is the challenge which many researchers and developers working in various relevant domains are facing. The difficulties here are coming from various sources, including the cost of making system dependable, the growing complexity of modern applications, their pervasiveness and openness, proliferation of computer-based systems into new emerging domains, wider reliance our society puts on these systems, complexity of ensuring the impact which various dependability means have on the resulting system dependability, difficulties in defining realistic and practical assumptions under which these means are to be applied, difficulties in setting dependability requirements and tracing them through all development phases, etc.

Dependability is an integrated concept encompassing a variety of attributes, including availability, reliability, safety, integrity, and maintainability. Four general means can be employed to attain dependability:<sup>2</sup> fault prevention, fault tolerance, fault removal, and fault forecasting. Clearly in practice one needs to apply a combination of all of these means to ensure the required dependability. It is important to understand that all these activities are centred around the concept of faults where possible faults are prevented or eliminated by using appropriate development and verification techniques, the remaining faults are tolerated at runtime to avoid system failures and estimated to help in predicting their consequences.

In this chapter, we follow the dependability terminology from<sup>2</sup> which introduces the following causal chain of dependability threads. It is said that the system failure to deliver its service is caused by an erroneous

system state, which, in its turn, is caused by a triggered fault. That means that faults can be silent for some time and that their triggering does not necessarily cause immediate failure. Errors are typically latent and the aim of fault tolerance is to detect them and deal with them and their causes before they make systems fail.

This book focuses on *fault tolerance means that are used to avoid system failures in the presence of faults*. The essence of fault tolerance<sup>3</sup> is in detecting errors and carrying the following system recovery. Generally speaking, during system recovery one needs to conduct two steps: error handling and fault handling.

*Error handling* can be conducted in one of the following three ways: backward error recovery (sometimes called rollback), forward error recovery (sometimes called rollforward) or compensation. Backward error recovery returns the system into a previous (assumed to be correct) state. The typical techniques are checkpoints, recovery points, recovery blocks, conversations, file backup, application restart, system reboot, transaction abort, etc. Forward error recovery moves the system into a new correct state, this type of recovery is typically carried out by employing exception handling techniques (found, for example, in many programming languages, such as Ada, Java, C++, etc.). Note that backward error recovery is usually interpreted as a particular case of forward error recovery. There have been considerable amount of work on defining exception handling mechanisms suitable for different domains, development and modelling paradigms, types of faults, execution environments, etc. (see, for example, recent book<sup>4</sup>). It is worth noting here that, generally speaking, the rollforward means are more general and run-time efficient than the rollback ones as they take advantage of the precise knowledge of the system erroneous state and move system into a correct state by using application-specific handlers. To conduct compensation one needs to ensure that the system contains enough redundancy to mask errors without interrupting the delivery of its service.

Various replication and software diversity techniques fall into this category as they mask the erroneous results without having to move the system into a state which is assumed to be correct. A wide range of software diversity mechanisms, including recovery blocks, conversations and N-version programming, has been developed and widely used in industry.

*Fault handling* activity has a nature which is very different from error handling as it intends to rid the system from faults to avoid new errors they may cause in the later execution. It starts with fault diagnosis, followed by isolation of the faulty component and system reconfiguration. After that

the system or its part needs to be re-initialized to continue to provide its service. Fault handling is usually much more expensive than error handling and is more difficult to apply as it typically requires some part of the system to be inactive to conduct reconfiguration.

Fault tolerance never comes for free as it always requires additional (redundant) resources which are employed in runtime for conducting detection and recovery. Specific fault tolerance mechanisms require various types of redundancy such as spare time, additional memory or disk space, extra exchange channels, additional code or messages, etc. Typically each scheme uses a combination of redundant resources, for example, simple retry always uses time redundancy, but may need extra disk space and code to save the checkpoints if we need to restore the system state before retrying.

The choice of the specific error detection, error handling and fault handling techniques to be used for a particular system is directly related to and depends upon the underlying fault assumptions. For example, replication techniques are typically used to tolerate hardware faults, whereas software diversity is employed to deal with software design bugs.

Let us now briefly discuss the main challenges in developing fault tolerant systems.<sup>5</sup> First of all, the fault tolerance means are difficult to develop or to use - this is because they increase system complexity by adding a new dimension to the reasoning about system behaviour. Their application requires a deep understanding of the intricate links between normal and abnormal behaviour and states of systems and components, as well as the state and behaviour during recovery. Secondly, fault tolerance (e.g. software diversity, rollback, exception handling) is costly as it always uses redundancy. Thirdly, system designers are typically reluctant to think about faults at the early phases of development. This results in making earlier decisions ignoring fault tolerance, which may make it more difficult or expensive to introduce fault tolerance at the later phases. More often than not, the developers fail to apply even the basic principles of software fault tolerance. For example, there is no focus on (i) a clear definition of the fault assumptions as the central step in designing any fault tolerant system, (ii) developing means for early error detection, (iii) application of recursive system structuring for error confinement, (iv) minimising and ensuring error confinement and error recovery areas, and (v) extending component specification with a concise or complete definition of failure modes. We can refer here to recent paper reporting a high number of mistakes made in handling exceptions in the C programs<sup>6</sup> and to the Interim Report on Causes of the August 14th 2003 Blackout in the US and Canada.<sup>7</sup> which clearly

shows that the problem was mostly caused by badly designed fault tolerance: poor diagnostics of faults, longer-than-estimated time for component recovery, failure to involve all necessary components in recovery, inconsistent system state after recovery, failures of alarm systems, etc. It is worth reminding here, as well, that the failure of the Ariane 5 launcher was caused by improper handling of an exception.<sup>8</sup>

All the factors above contribute to the fact that a substantial part of system failures are caused by mistakes in fault tolerance means.<sup>1</sup> We believe that a closer synergy between software engineering phases, methods and tools and fault tolerance will help alleviating such current problems.

### 3. Defining Software Engineering

Software engineering (SE) is a quite new field of Computer Science, recognized in the 1968 NATO conference in Garmisch (Germany) as an emergent discipline. Today, many different definitions of software engineering have been proposed, trying to explain its main characteristics:

- “Application of *systematic, disciplined, quantifiable approach* to the *development, operation, and maintenance* of software”;<sup>9</sup>
- “Software engineers should adopt a *systematic and organised approach* to their work and use appropriate tools and techniques depending on the problem to be solved, the development constraints and the resources available”;<sup>10</sup>
- “Software Engineering is the field of computer science that deals with the building of software systems that are *so large or so complex* that they are built by a team or teams of engineers”;<sup>11</sup>
- “Software engineering is the branch of systems engineering concerned with the development of *large and complex software intensive systems*”... “It is also concerned with the *processes, methods and tools* for the development of software intensive systems in an *economic and timely manner*”.<sup>12</sup>

Most of those well known definitions point out different characteristics or perspectives to be used when looking at the software engineering discipline. Some examples, taken from experienced computer scientists made in the last forty years, will help to identify the key points common to most of the definitions of SE, and help us to illustrate why and when the software engineering discipline is needed. The *i*) Ariane 5, the Therac-25 radiation therapy machine, the Denver Airport (and others) big software failures,<sup>13</sup>

and the *ii*) on-board shuttle group excellence<sup>14</sup> examples will be used for this purpose.

### ***3.1. If Software Fails, It May Cost Millions of Dollars and Injure People***

As already pointed in Section 1, software is pervasive (it is everywhere around us, even if we do not see it), it controls many devices used everyday, and more and more critical systems (i.e., those systems whose malfunctioning can injure people or create high economic losses).

The Ariane 5, X-ray machine and Denver Airport are some examples of critical systems which, due to software systems malfunctioning, ended up being big catastrophic failures<sup>a</sup>. The Ariane 5 shuttle, launched on June 4th 1996, broke up and exploded forty seconds after initiation of the flight sequence, due to a software problem. People were killed. The Therac-25 radiation-treatment machine for cancer therapy injured and even killed several patients by administering a radiation overdose. The Denver Airport software, responsible for controlling 35 kilometers of rails and 4000 tele-wagons never worked properly and after 10 years of recurring failures it has been recently dismissed:<sup>15</sup> millions of dollars were wasted. In all three cases, the main causes of failure were undisciplined management of requirements, imprecise and ambiguous communication, instable architectures, high complexities, incoherence among requirements design and implementation, low automation, and insufficient verification and validation.

### ***3.2. How to Make Good Software***

While the previous examples described what might happen when software engineering techniques are not taken into consideration, the on-board shuttle group example of excellence (taken from the 1996 white-paper written by Fishman<sup>14</sup>) shows results that can be achieved when software engineering best practices are applied in practice. It describes how the software governing a 120-ton space shuttle is conceived: such a software system is composed of around 500.000 lines of code, it controls 4 billion dollars worth of equipment, and decides the lives of a half-dozen astronauts. What makes this software and their creators so extraordinary, is that it never crashes and is bug free (according to<sup>14</sup>). The last three versions of the program

---

<sup>a</sup>Those and many other examples of catastrophic failures are described in .<sup>13</sup>

(each one of 420,000 lines of code) had just one fault each. The last 11 versions of this software system had a total of 17 faults. Commercial programs of equivalent complexity would have 5,000 faults. How did the developers achieve such high software quality?

It was simply the result of applying most of the Software Engineering (SE) best practices:

- *SE allows for a disciplined, systematic, and quantifiable development:* The on-board shuttle group is the antithesis of the up-all-night, pizza-and-roller-hockey software coders who have captured the public imagination. To be this good, the on-board shuttle group is very ordinary – indistinguishable, focused, disciplined, and methodically managed creative enterprise;
- *SE does not only concern programming:* Another important factor discussed in the Fishman's report<sup>14</sup> is that about one-third of the process of writing software happens before anyone writes a line of code. Every critical requirement is documented. Nothing in the specification is changed without agreement and understanding. No coder changes a single line of code without carefully outlining the change;
- *SE takes into consideration maintenance and evolution:* As explicitly stated in<sup>9</sup> maintenance and evolution are important factors when engineering software systems. They allow system evolution, while limiting newly introduced faults;
- *SE is for mid to large systems:* Applying SE practices is indeed expensive and requires effort. While non critical, small systems can require just a few SE principles, applying the best SE practices for the development of critical, large software systems is mandatory;
- *Development cost and time are key issues:* The main success of this example is not the software but the software process the team uses. Recently, much effort has been spent on identifying new software processes (like the Unified Software Process<sup>16</sup>), and software maturity frameworks which allows the improvement of the software development process (like the Capability Maturity Model – CMM<sup>17</sup> or the Personal Software Process – PSP<sup>18</sup>). Nowadays software processes take explicitly into consideration tasks like managing groups, setting deadlines, checking the system cost to stay on budget, and to deliver software which respects the expected qualities.

#### 4. Fault Tolerance Engineering: from Requirements to Code

Initial solutions relegated fault tolerance (and specifically, exception handling) very late during the design and implementation phases of the software life-cycle. More recently, instead, the need of explicit exception handling mechanisms during the entire life cycle has been advocated by some researchers as one of the main approaches to ensure the overall system dependability.<sup>19,20</sup>

In particular, it has been recognized that different classes of faults, errors and failures can be identified during different phases of software development. A number of studies have been conducted so far aiming to understand where and how fault tolerance can be integrated in the software life-cycle.

In the remaining part of Section 4 we will show how fault tolerance has been recently addressed at the different phases of the software process. The phases that will be taken into consideration are requirements, high-level (architectural) design, and low-level design thus reflecting current study on fault tolerance techniques during such phases.

##### 4.1. Requirements Engineering and Fault Tolerance

Requirements Engineering is concerned with identifying the purpose of a software system, and the contexts in which it will be used. Different theories and methodologies for finding out, modelling, analyzing, evolving and checking software system requirements<sup>21</sup> have been proposed so far.

Being requirements the first artefacts produced during the software process, it is important to document expected faults and ways to tolerate them. Some approaches have been proposed for this purpose, the most known being analyzed in<sup>20,22,23</sup> and subsequent work.

In<sup>22,24,24</sup> the authors describe a process for systematically investigating exceptional situations at the requirements level and provide an extension to standard UML use case diagrams in order to specify exceptional behaviour. In<sup>20</sup> it is described how exceptional behaviours can be specified at the requirements level, and how those requirements can drive component-based specification and design according to the Catalysis process. In<sup>23</sup> an approach for analyzing the safety and reliability of requirements based on use cases is proposed: normal use cases are extended with exceptional use cases according to,<sup>22</sup> then use cases are annotated with their probability of success and successively translated into Dependability Assessment Charts, eventually used for dependability analysis.

#### 4.2. Software Architecture and Fault Tolerance

Software Architecture (SA) has been largely accepted as a well suited concept to achieve a better software quality while reducing the time and cost of production. In particular, a software architecture specification<sup>25</sup> represents the first, in the development life-cycle, complete system description. It provides both a high-level behavioural abstraction of components and of their interactions (connectors) and, a description of the static structure of the system.

Typical SA specifications model only *normal* behaviour of the system, while ignoring *abnormal* ones. As a consequence, the system may fail in unexpected ways due to some faults. In the context of critical systems with fault tolerance requirements it becomes necessary to introduce fault tolerance information at the software architecture level. In fact, the error recovery effectiveness is dramatically reduced when fault tolerance is commissioned late in the software life-cycle.<sup>19</sup>

Many approaches have been proposed for modelling and analyzing fault tolerant software architectures. While a comprehensive survey on this topic is described in,<sup>26</sup> this introductory section simply identifies the main topics covered by the existing approaches, while providing some references to the existing work.

- Fault Tolerant SA specification: as discussed in many papers (e.g.,<sup>27-29</sup>) a software architecture can be specified using box-and-line notations, formal architecture description languages (ADLs) or UML-based notations. As far as the specification of fault tolerant software architectures is concern, both formal and UML-based notations have been used. The approaches proposed in<sup>30-32</sup> are examples of formal specifications for Fault Tolerant SA: traditional architecture description languages are usually extended in order to explicitly specify error and fault handling. The approaches in, e.g.,<sup>20,33,34</sup> use UML-based notations for modelling Fault Tolerant SA: new UML profiles are created in order to be able to specify fault tolerance concepts;
- Fault Tolerant SA analysis: analysis techniques (such as deadlock detection, testing, checking, simulation, performance) allow software engineers to assess the software architecture and to evaluate its quality with respect to expected requirements. Some approaches have been proposed for analyzing Fault Tolerant SA: most of them check the architectural model conformance to fault tolerant require-

ments or constraints (like in<sup>35,36</sup>). A testing technique for Fault Tolerant SA is presented in;<sup>34</sup>

- Fault Tolerant SA styles: according to,<sup>37</sup> an architectural style is “*a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints on the way the composition is done*”. Many architectural styles have been proposed for Fault Tolerant SA: the idealized fault tolerant style (in<sup>34</sup>), the iC2C style (which integrates the C2 architectural style with the idealized fault tolerant component style<sup>30</sup>), the idealized fault tolerant component/connector style;<sup>38</sup>
- Fault Tolerant SA middleware support: when coding software architecture via component-based systems, middleware technology can be used to implement connectors, coordination policies and many other features. In<sup>39</sup> a CORBA implementation of the proposed architectural exception handling is proposed. In<sup>40</sup> the authors propose an approach for exception handling in component composition at the architectural level with the support of middleware. Many projects have been conducted to provide fault tolerance to CORBA applications, like AQuA, Eternal, IRL, and OGS (see<sup>41</sup>). More details about middlewares are given in Section 6.

#### 4.3. Low-level Design and Fault Tolerance

The low-level design phase (hereafter simply called “design”) takes as input information collected during the requirement and architecting phases and produces a design artefact to be used by developers for guiding and documenting software coding. When dealing with fault tolerant systems, the design phase needs to benefit from some clear and domain specific tools and methodologies to drive the implementation of a particular fault tolerant technique. This introductory section will present only such approaches that treat fault tolerant during the software development process (from requirements to code).

In<sup>42,43</sup> two approaches are presented for transiting from architectural design to low-level design through the definition of fault tolerant design patterns. In<sup>33</sup> an initial study on the CORRECT MDA approach is introduced: given a coordinated atomic action specification, the approach enables the automatic production of Java code. This approach has been successively refined in other papers, and recently presented in.<sup>44</sup> In<sup>20</sup> an approach for fault tolerance specification and analysis during the entire development process is

proposed. It considers how to specify normal and exceptional requirements, how to use this information for driving the component specification and design phase, and how to implement all such information using a Java-based framework Java. The proposed software process is based on the Catalysis process. In<sup>34</sup> a similar strategy is adopted based on the UML Components Process (even if this paper is more towards testing).

## 5. Verification and Validation of Fault Tolerant Systems

Fault tolerance techniques alone are not enough to achieve full dependability, since unexpected faults cannot be always avoided nor tolerated.<sup>45</sup> In addition it is important to note that fault tolerant systems inevitably contain faults. Verification and validation (V&V) techniques are demonstrated successful means to assure that expected properties and requirements are satisfied in system models and implementation.<sup>34</sup> In this setting V&V techniques are the solution for removing faults from the system.

Furthermore, V&V should be used at each different life-cycle phase since fault tolerance engineers the entire software development life-cycle.

Different classes of faults, errors, and failures must be identified and dealt with at each phase of software development, depending on the abstraction level used in modeling the software system under development. Thus, each abstraction level requires specific design models, implementation schemes, verification techniques, and verification environments.

Verification and validation techniques aim to ensure the correctness of a software system or at least to reduce the number or severity of faults both during the development and after the deployment of a system. There are two different class of verification methods, exhaustive methods that conduct an exhaustive exploration of all possible behaviours and non-exhaustive methods that explore only some of the possible behaviours. In the exhaustive class there are model checking, theorem provers, term rewriting systems, proof checker systems, and constraint solvers. In the non-exhaustive class there are testing and simulation, the veteran techniques, commonly and widely used but that can easily miss significant errors when verifying complex and huge systems. In literature several approaches have been proposed in the last years trying to apply V&V techniques to fault-tolerant systems and they are surveyed in the following subsections. In particular, Section 5.1 reports the use of model checking techniques for fault-tolerant systems, Section 5.2 summarizes experiences with theorem provers applied to fault tolerant systems, Section 5.3 shows approaches that exploit constraint solvers for verifying fault tolerant systems, and finally Section 5.4 reports how

fault tolerance has been complemented with testing techniques. Furthermore, with the introduction of UML<sup>46</sup> as the de-facto standard to model software systems and its widespread adoption in industrial contexts, many approaches have been proposed to use UML for modeling and evaluating dependable systems (e.g.,<sup>47–50</sup>); they are reported in Section 5.5.

### 5.1. *Model Checking*

Model checkers take as input a formal model of the system, typically described by means of state machines or transition systems, and verify if it satisfies temporal logic properties.<sup>51</sup>

Several approaches have been proposed in the last years focusing on model checking of fault tolerant systems, such as.<sup>52–54</sup>

These papers describe approaches and show their application to real case studies testifying that model checking is a promising and successful verification technique. First of all model checking techniques are supported by tools, which facilitates their application. Secondly, in case the verification detects a violation of a desired property, a counter example showing how the system reaches the erroneous state in which the property is violated is produced.

Model checking approaches for fault tolerant systems typically require to specify normal behaviours, failing behaviours, and fault recovering procedures. Thus, fault tolerant systems are subjected to the state explosion problem that afflicts model checkers also in verifying systems that do not consider exceptional behaviours.

One approach that can be used for avoiding the state explosion problem is the partial model checking technique introduced in.<sup>55</sup> This technique that tries to gradually remove parts of the system is successfully applied for security analysis and an attempt to use it for fault tolerant systems is in.<sup>56</sup>

### 5.2. *Theorem Provers*

Interactive theorem provers start with axioms and try to produce new inference steps using rules of inference. They require a human user to give hints to the system. Working on hard problems usually requires a skilled user. A logic characterization of fault tolerance is given in,<sup>56</sup> while approaches that apply theorem prover techniques to fault tolerant systems are in.<sup>57–60</sup>

### 5.3. *Constraint Solvers*

Given a logical formula, expressed in a suitable logic, constraint solvers attempt to find a model that makes the formula true. The model typically is a match between variables and values. One of the most famous constraint solvers, based on first-order logic, is Alloy Analyzer, the verification engine of Alloy.<sup>61</sup> In,<sup>62</sup> authors propose an approach that exploits Alloy for modeling and formally verifying fault-tolerant distributed systems. More precisely they focus on systems that use exception handling as mechanism for fault tolerance and in particular they consider systems designed by using Coordinated Atomic Actions (CAA).<sup>63</sup> CAA is a fault-tolerant mechanism that uses concurrent exception handling and unifies the features of two complementary concepts: conversation and transaction. Conversation<sup>64</sup> is a fault-tolerant technique for performing coordinated error recovery in a set of participants that have been designed to interact with each other to provide a specific service (cooperative concurrency).

### 5.4. *Testing*

Testing refers to the dynamic verification of a system's behaviour based on the observation of a selected set of controlled executions, or test cases.<sup>65</sup> Testing is the main fault removal technique.

A real world project involving 34 independent programming teams for developing program versions of an industry-scale avionics application is presented in.<sup>66</sup> Detailed experimentations are reported to study the nature, source, type, delectability, and effect of faults uncovered in the programming versions. A new test generation technique is also presented with an evaluation of its effectiveness.

Another approach<sup>67</sup> shows how fault tolerance and testing can be used to validate component-based systems. Fault tolerance requirements guide the construction of a fault-tolerant architecture, which is successively validated with respect to requirements and submitted to testing.

### 5.5. *UML-based approaches for modeling and validating dependable systems*

The approaches considered in this section share the idea of translating design models into reliability models.

A lot of works and solutions have been proposed in the context of the European ESPRIT project HIDE.<sup>49</sup> This project aims at the creation of an integrated environment for designing and verifying dependable systems

modeled in UML. In<sup>48</sup> authors propose automatic transformations from UML specifications, augmented with additional required information (i.e., fault occurrence rate, percentage of permanent faults, etc...), to Petri Net Models.

A modular and hierarchical approach for dependable software architectures is proposed in.<sup>50</sup> The language used for describing software architectures is UML. The approach suggests a refinement process allowing the description of critical parts of the model when information becomes available in the following design phases.

In<sup>47</sup> authors convert UML models to dynamic fault trees. In this work UML is mainly used as a language for describing module substitution and error propagation.

## 6. Languages and Frameworks

It is of great importance that engineers could find in their development tools features that help them to deal with the increase of the complexity because of the incorporation of fault-tolerant software techniques into the final software. Each development tool studied in this section helps in separating the code to implement the software system function (as described by its functional specification) from the necessary code to implement the service restoration (or simply “recovery”), when a deviation from the correct service was detected (by the implemented error detection technique, of course). The choice of recovery features depends on the classes of faults that to be tolerated. For example, transient faults, which are the faults that eventually disappear without any apparent intervention, can be tolerated by error handling.

Each of the tools that is presented below, allows engineers to solve this issue and sometimes with several possibilities. The selection amongst all the proposed solution paths will depend on: costs in terms of money, processing power (performance), and memory size. But also on the costs (quantifiable or not) induced by the failure of the software system, which is the most important one to evaluate precisely in order to decide on the requirements for fault tolerance.

This section addresses three types of development environments: programming languages, fault-tolerant frameworks and advanced fault-tolerant frameworks. The choice among these three categories will depend on the complexity of the fault tolerance requirements.

### 6.1. Programming Languages Perspectives

Some programming languages incorporate fault tolerance techniques directly as part of their syntax or indirectly by features that allow engineers to implement them. One reason for having fault tolerance support at the programming language level is due to the increased performance as consequence of the application-specific knowledge. Another reason is to offer to programmers that need to comply to standard programming languages the capability to design and develop more easily fault-tolerant applications.

#### 6.1.1. Exception Handling

As stated in the previous section, one of the features to be provided by a fault tolerance technique is to support separation of the fault tolerance instructions (for recovery objectives) from the rest of the software and to activate them automatically, when necessary. The obvious moment to activate the recovery behaviour is when it is impossible to finish the operation that the software is carrying out. An exception is exactly defined as the notification of the impossibility of finishing an operation and it can be used to know that the software is going to fail if no action is taken. It points out that the period of time between the notification of the exception and the failure of the software can be used to apply the fault-tolerant instructions in order to keep the software running and avoid a failure. This is called *Exception handling* (EH). It is the most popular way used by modern software and it plays a vital part in the implementation of fault tolerance in software system.

Nowadays, various exception handling models are part of practical programming languages like Ada, C++, Eiffel, Java, ML and Smalltalk. Almost all languages have similar basic types of exceptions and similar constructs to introduce new exception types and to handle exceptions (e.g. *try/throw/catch* in Java). Thus, exception handling is a good technique to implement fault-tolerant sequential programs.

In the context of distributed concurrent software (network of computing nodes), the situation is different. The exception handling must be defined according to the semantics of concurrency and distribution. ERLANG<sup>68</sup> is a declarative language for programming concurrent and distributed software systems with EH features. This language has primitives which support the creation of processes (separated unit of computation), the communication between processes over a network of nodes and the handling of errors when a failure caused a process to terminate abnormally. The statement spawn

allows a process to create a new process on a remote node. Whenever a new process is created, the new process will belong to the same process group as the process that evaluated the spawn statement. Once a process terminates (normally or abnormally) its execution, a special signal is sent to all processes which belong to the same group. The value of one of the parameters that compose the signal is used to detect if the process terminated abnormally. In order to avoid propagating an abnormal signal to the other processes of the group (i.e. to ensure failure containment), the default behaviour must be changed. This is achieved by using the catch instruction, which defines a scope to deal with errors occurred on the monitored expression.

### 6.1.2. *Atomic Actions*

Fault tolerance in distributed concurrent software systems can also be achieved using the general concept of atomic actions. A group of components (participants, threads, processes, objects, etc.) that cooperate to achieve a joint goal, without information flow between the group and the rest of the system for the period necessary to achieve the goal, constitutes an atomic action. These components are designed to cooperate inside the action, so that they share work and exchange information in order to complete the action successfully. Atomicity guarantees that if the action is successfully executed, then its results and modifications on shared data become visible to other actions. But if an error is detected, all the components take part in a cooperative recovery in order to return without changes on the shared data. These scheme characteristics allows the containment and recovery to be easily achieved since the error detection, propagation and recovery all occur within a single atomic action. Therefore fault tolerance steps can be attached to each of the atomic action that forms part of the software, independently from each other. The first fault-tolerant atomic action scheme proposed was the conversation scheme.<sup>64</sup> It allows tolerating design faults by making use of software diversity and participant rollback. Other schemes including fault tolerance has been proposed since then and developed as part of programming languages. For example, Avalon,<sup>69</sup> takes advantage of inheritance to implement atomic actions in distributed object-oriented applications. Avalon relies on the Camelot<sup>70</sup> system to handle operative-system level details. Much of the Avalon design has been inspired by Argus.<sup>71</sup>

### 6.1.3. *Reflection and Aspect-Orientation*

Other software technology considered for handling software faults and which is related to programming languages is “reflection”.<sup>72</sup> Reflection is the ability of a computational system to observe its own execution and, as a result of that observance, perhaps make changes to that execution. Software based on reflective facilities is structured into different levels: the base level and one or more metalevels. Everything in the implementation and application (the syntax, the semantics, and the run-time data structures) is “open” to the programmer for modification via the metalevels.<sup>73</sup> The metalevels can be used to handle the fault tolerance strategies. Therefore, this layer structure allows programmers to separate the recovery steps (part of the metalevels) from necessary steps to reach the functional goal (part of the base level). The fact that metalevels can observe the base level computation allows halting its execution when any deviation is observed (according to some parameter of reference) to start the recovery.

A generic solution to implement fault tolerance that is or can be chosen for many programming languages consists in:

- extending the programming language with non-standard constructs and semantics;
- extending the implementation environment underlying the programming language to provide the functionality, but with an interface expressed using existing language constructs and semantics;
- extending the language with specific abstractions, implemented with existing language constructs and semantics (e.g., abstract data types intended to support software fault tolerance) perhaps expressed in well-recognised design patterns;
- or combinations of the previous approaches.

Aspect-orientation has been accepted as a powerful technique for modularizing crosscutting concerns during software development in so-called aspects. Similar to reflection, aspect-oriented techniques provide means to extend a base programs with additional state, and define additional behaviour that is to be triggered at well-defined points during the execution of the program. Experience has shown that aspect-oriented programming is successful in modularizing even very application-independent, general concerns such as distribution and concurrency, and examples of using aspect-orientation to achieve fault tolerance are given in.<sup>74-78</sup>

Nevertheless, if complex fault tolerance requirements are given to your application, choosing programming languages will be risky and the selection

of a framework will then be necessary.

### 6.2. *Frameworks for Fault Tolerance*

According to<sup>79</sup> “*a framework is a reusable design expressed as a set of abstract classes and the way their instances collaborate. It is a reusable design for all or part of a software system. By definition, a framework is an object-oriented design. It doesn't have to be implemented in an object-oriented language, though it usually is. Large-scale reuse of object-oriented libraries requires frameworks. The framework provides a context for the components in the library to be reused.*”

CORBA (Common Object Request Broker Architecture), is a good example of a framework, which was conceived to provide application interoperability. Unfortunately, CORBA and other traditional frameworks cannot often meet the demanding quality of service (QoS) requirements (including the fault-tolerance ones) for certain specialised applications. This is why these frameworks are often extended to include fault tolerance techniques in order to become predictable and reliable. This is done in FT CORBA specification.<sup>80</sup> It defines architecture, a set of services, and associated fault tolerance mechanisms that constitute a framework for resilient, highly-available, distributed software systems. Fault tolerance is achieved by features that allow designers to replicate objects in a transparent way. The set of several replicas for a specific object defines an object group. However, a client object is not aware that the server object is replicated (server object group). Therefore, the client object invokes methods on the server object group, and the members of the server object group execute the methods and return their responses to the client, just like a conventional object.

The same approach has also been followed at the higher levels of abstraction. For example, the well-known coordination language Linda,<sup>81</sup> which has been extended to facilitate programming of fault-tolerant parallel applications. FT-Linda,<sup>82</sup> for instance, is an extension of the original Linda model, which defines new concepts as stable tuple spaces (stable TSs), failure tuple and new syntax to achieve atomic execution of a series of TSs operations. A stable TS represents a tuple that survives to a failure. This tuple stability is achieved by replicating the tuple on the multiple hosts that conform the distributed environment where the software is deployed. FT-Linda uses monitoring to detect failures. The main type of failure that is addressed by FT-Linda corresponds to host failure, which means that the host has been silent for longer than a pre-defined interval. When such type of failure is

detected, the framework automatically notifies all processes by signalling a failure tuple in a stable TS available to them. Notice, there must exist a specific process in the software in charge of watching for a failure tuple and starting the corresponding recovery process. Atomic execution is denoted by angle brackets and represents the all-or-none execution of the group of tuple space operations enclosed by the angle brackets. There are also some language primitives that allow tuples to be moved or copied from one TS to another one.

Other extension of Linda are oriented towards mobile applications using the agent paradigm, is CAMA.<sup>83</sup> It brings fault tolerance to mobile agent applications, by the use of a novel exception handling mechanisms developed for this type of applications. This exception handling mechanism consists in attaching application-specified handlers to agents to achieve recovery. Therefore, to the set of primitives derived from Linda (e.g. create, delete, put, get, etc.), CAMA adds some primitives to catch and raise inter-agent exceptions (e.g. raise, check, wait).

Linda has strongly influenced the JavaSpaces<sup>84</sup> system design. They are similar in the sense that they store collections of information (resources) which can be later searchable by value-based lookup in order to allow members of distributed software systems to exchange information. JavaSpaces is part of Jini Network Technology,<sup>85</sup> which is an open architecture that enables developers to create network-centric service. Jini has a basic mechanism used for fault-tolerant resource control, which is referred to as Lease. This mechanism is used to define a holding interval of time on a resource by the party that requests access on such resource. The mechanisms notifies (error detection) when the leases expires, so that actions (recovery) on lease expiration can be taken for the requestor party.

Even though these framework extensions represent good tools to implement fault-tolerance they are still “extensions” of existing tools. The next section will present frameworks that have been designed with the central objective to support the design and implementation of fault tolerance.

### **6.3. Advanced Frameworks for Fault Tolerance**

Frameworks that have been defined to allow designers/programmers to develop fault-tolerant software by the implementation of fault tolerance techniques share, in general, the following characteristics:<sup>86</sup>

- many details of their implementation are made transparent to the programmers;

- they provide well-defined interfaces for the definition and implementation of fault tolerance techniques;
- they are recursive in nature (each component can be seen as a system itself).

Arjuna<sup>87</sup> programmed in C++ and Java, is one of those. Arjuna permits the construction of reliable distributed applications in a relatively transparent manner. Reliability is achieved through the provision of traditional atomic transaction mechanisms implemented using only standard language features. It provides basic capabilities to handle recovery, persistence and concurrency control to the programmer, while at the same time giving flexibility to the software by allowing those capabilities to be refined as required by the demands of the application.

Other similar work is OPTIMA,<sup>88</sup> which is an object-oriented framework (developed for Ada, Java and AspectJ) that provides the necessary runtime support for the “Open Multithreaded Transactions” (OMT) model. OMT is a transaction model that provides features for controlling and structuring not only accesses to objects, as usually happens in transaction systems, but also threads taking part in a same transaction in order to perform a joint activity. The framework provides features to fork and terminate threads inside a transaction, but restricting their behaviours in order to guarantee correctness of transaction nesting and isolation among transactions.

The DRIP framework<sup>89</sup> provides a technological support to implement DIP<sup>90</sup> models in Java. DIP combines Multiparty Interactions<sup>91,92</sup> and Exception Handling<sup>93</sup> in order to support design dependable interactions among several processes. As an extension of DRIP, the CAA-DRIP implementation framework<sup>44</sup> provides a way to execute Java programs that have been designed using the COALA design language<sup>94</sup> that exploits the Coordinated Atomic actions (CA actions) concept.<sup>95</sup>

Other advanced frameworks for fault-tolerance are currently being designed. For example, the MetaSolve design framework, supporting dynamic selection and parallel composition of services, has been proposed for developing dependable systems.<sup>96</sup> This approach defines an architectural model, which makes use of service-oriented architecture features to implement fault tolerance techniques based on meta-data. Furthermore, the software implemented using the MetaSolve approach will be able to adapt itself at run-time in order to provide dynamic fault-tolerance. The ability of such software to dynamically respond to potentially damaging changes by adaptation in order to maintain an acceptable level of service is referred to as

dynamic resilience. The architecture of software that follows this approach relies on dynamic information about software components in order to make decisions for dynamic reconfiguration. Such metadata then will be used in accordance with some resilient policies ensuring that the desirable dependable requirements are met.

## 7. Contribution of this Book to the Topic

This book will contribute to the overall topic of Software Engineering and Fault Tolerance with nine papers, briefly described in the following, and categorized according to the three parts identified at the beginning of this paper:

- Part A: *Fault tolerance engineering: from requirements to code*
  - In “*Exploiting Reflection to Enable Scalable and Performant Database Replication at the Middleware Level*” Jorge Salas, Ricardo Jimenez-Peris, Marta Patino-Martinez and Bettina Kemme introduce a design pattern for data base replication using reflection at interface level. It permits a clear separation between regular function and replication logic. This design pattern allows to obtain good performance and scalability properties.
  - In “*Adding Fault-Tolerance to State Machine-Based Designs*”, Sandeep S. Kulkarni, Anish Arora and Ali Ebnenasir present a non application specific approach for automatic re-engineering of code in order to make it fault-tolerant and safety. This generic approach is using model based transformations of programs that must be atomic concerning accesses to variables.
  - In “*Replication in Service-Oriented Systems*”, Johannes Osrael, Lorenz Froihofer and Karl M. Goeschka present a state of the art of replication protocols and of replication in service oriented architectures supported by middleware’s. Then is shows how to enhance to existing solutions provided in the service oriented field with already known designs for replication in traditional systems.
- Part B: *Verification and validation of fault tolerant systems*
  - In “*Embedded Software Validation Using On-Chip Debugging Mechanisms*”, Juan Pardo, José Carlos Campelo, Juan Carlos Ruiz and Pedro Gil present how to practically use on-chip debugging to perform fault-injection in a non-intrusive way. This

portable approach offers a verification and validation mean for checking and validating the robustness of COTS based embedded systems.

- In “*Error Detection in Control Flow of Event-Driven State Based Applications*”, *Gergely Pinter and Istvan Majzik* present a formal approach using state-charts to detect two classes of faults: the one made during state-chart refinement (using temporal logic model-checking) and the one made at implementation (using model based testing).
- In “*Fault-Tolerant Communication for Distributed Embedded Systems*”, *Christian Kühnel and Maria Spichkova* present a formal specification using the FOCUS formal framework of FlexRay and FTCom. It thus provides a precise semantics useful for analyzing dependencies and for the verification (using Isabelle/HOL) of existing implementations.
- Part C: *Languages and Tools for engineering fault tolerant systems*
  - In “*A Model Driven Exception Management Framework*”, *Susan Entwistle and Elizabeth Kendall* present a model driven engineering approach to the engineering of fault-tolerant systems. An iterative development process using the UML 2 modelling language and model transformations is proposed. The engineering framework proposes generic transformations for exceptions handling strategies thus raising the exception handling at higher level of abstraction than only implementation.
  - In “*Runtime Failure Detection and Adaptive Repair for Fault-Tolerant Component-Based Applications*”, *Rong Su, Michel Chaudron and Johan Lukkien* present formally a fault management mechanism useful for systems designed using a component model. Run-time failure are detected and the repair strategy is selected using a rule-based approach. An adaptive technique is proposed to enhance progressively the selection of the repair strategy. A forthcoming development framework, Robocop, will provide an implementation of this mechanism.
  - In “*Extending the Applicability of the Neko Framework for the Qualitative and Quantitative Validation and Verification of Distributed Algorithms*”, *Lorenzo Falai and Andrea Bondavalli* present a development framework written in Java allowing rapid prototyping of Java distributed algorithms. An import function allows for direct integration of C and C++

programs via glue-code. The framework offers some techniques for qualitative analysis that can be used specifically for the fault tolerance parts of the distributed program developed with the framework.

### Acknowledgments

The book editors wish to thank Andrea Bondavalli and Rogerio de Lemos for their constructive comments on this introductory chapter and Alfredo Capozucca and Joerg Kienzle for their comments and contribution to Section 6.

### References

1. A. Romanovsky, A Looming Fault Tolerance Software Crisis?, in *2006 NATO Workshop on Building Robust Systems with Fallible Construction (also CS-TR-991 Newcastle University, UK)*, 2006.
2. A. Avizienis, J.-C. Laprie, B. Randell and C. E. Landwehr, Basic Concepts and Taxonomy of Dependable and Secure Computing, *IEEE Trans. Dependable and Sec. Comput. vol. 1, numb. 1*, 2004.
3. P. Lee and T. Anderson, *Fault Tolerance: Principles and Practice, Second Edition* (Prentice-Hall, 1990).
4. C. Dony, J. L. Knudsen, A. Romanovsky and A. Tripathi, *Advances in Exception Handling Techniques*, LNCS-4119 edn. (Springer-Verlag, 2006).
5. A. Romanovsky, Fault Tolerance through Exception Handling in Ambient and Pervasive Systems, in *SBES 2005 - 19th Brazilian Symposium on Software Engineering, Brazil*, October 6, 2005.
6. M. Bruntink, A. van Deursen and T. Tourwe, Discovering faults in idiom-based exception handling, in *ICSE 2006: Proceedings of the International Conference on Software Engineering*, (ACM Press, Shanghai, China, 2006).
7. The U.S. Secretary of Energy and the Minister of Natural Resources Canada, *Interim Report: Causes of the August 14th Blackout in the United States and Canada*. Canada U.S. Power System Outage Task Force, 2003.
8. J.-L. Lions, *Ariane 5 flight 501 failure. Technical report* (ESA/CNES, 1996).
9. Recommended Practice for Architectural Description of Software-Intensive Systems, in *The Institute of Electrical and Electronics Engineers (IEEE) Standards Board*, (IEEE-Std-1471-2000, September 2000)
10. I. Sommerville, *Software engineering (8th ed.)* (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006).
11. C. Ghezzi, M. Jazayeri and D. Mandrioli, *Fundamentals of Software Engineering*, second edn. (Prentice Hall, September 2002).
12. W. Emmerich, *Engineering Distributed Objects* (John Wiley and Sons Ltd, 2000).
13. I. Peterson, *Fatal Defect: Chasing Killer Computer Bugs* (Vintage, 1996).

14. C. Fishman, They Write the Right Stuff (in the FastCompany magazine, Issue 06, Dec 1996/Jan 1997, Page 95).
15. D. Jackson, Dependable Software by Design *Scientific American* June 2006.
16. I. Jacobson, G. Booch and J. Rumbaugh, *The Unified Software Development Process* (Addison Wesley, Object Technology Series, 1999).
17. The capability maturity model. <http://www.sei.cmu.edu/cmm/>.
18. The personal software process. <http://www.sei.cmu.edu/tsp/psp.html>.
19. R. de Lemos and A. Romanovsky, Exception Handling in the Software Life-cycle *International Journal of Computer Systems Science and Engineering* **vol. 16, numb. 2** (CRL Publishing, March 2001).
20. C. M. F. Rubira, R. de Lemos, G. R. M. Ferreira and F. C. Filho, Exception handling in the development of dependable component-based systems *Softw. Pract. Exper.* **vol. 35, numb. 3** (John Wiley & Sons, Inc., New York, NY, USA, 2005).
21. B. Nuseibeh and S. Easterbrook, Requirements Engineering: A Roadmap, in *ACM ICSE 2000, The Future of Software Engineering*, ed. A. Finkelstein, year 2000.
22. A. Shui, S. Mustafiz, J. Kienzle and C. Dony, Exceptional Use Cases, in *MoDELS*, 2005.
23. S. Mustafiz, X. Sun, J. Kienzle and H. Vangheluwe, Model-driven assessment of use cases for dependable systems, in *MoDELS*, 2006.
24. A. Shui, S. Mustafiz and J. Kienzle, Exception-aware requirements elicitation with use cases *Advanced Topics in Exception Handling Techniques, Springer, Berlin* 2006.
25. D. E. Perry and A. L. Wolf, Foundations for the study of software architecture, in *SIGSOFT Software Engineering Notes*, Oct 1992.
26. H. Muccini and A. Romanovsky, Architecting Fault Tolerant Systems: a Comparison Framework *University of L'Aquila Technical Report* 2007.
27. N. Medvidovic and R. N. Taylor, A Classification and Comparison Framework for Software Architecture Description Languages *IEEE Transactions on Software Engineering* **vol. 26, numb. 1**, January 2000.
28. N. Medvidovic, D. S. Rosenblum, D. F. Redmiles and J. E. Robbins, Modeling Software Architectures in the Unified Modeling Language *ACM Transactions on Software Engineering and Methodology (TOSEM)* **vol. 11, numb. 1**, January 2002.
29. D. Garlan, Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events, in *Formal Methods for Software Architectures*, (Lecture Note in Computer Science, 2804, 2003).
30. F. C. Filho, P. A. de C. Guerra and C. M. Rubira, An Architectural-Level Exception-Handling System for Component-Based Applications, in *LADC03*, 2003.
31. F. C. Filho, P. H. S. Brito and C. M. F. Rubira, A framework for analyzing exception flow in software architectures, in *ICSE 2005 Workshop on Architecting Dependable Systems (WADS05)*, 2005.
32. C. Gacek and R. de Lemos, *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective* (Springer-Verlag, 2006), ch. Ar-

chitectural Description of Dependable Software Systems, pp. 127–142.

- 33. N. Guelfi, R. Razavi, A. Romanovsky and S. Vandenberghe, DRIP Catalyst: An MDE/MDA Method for Fault-tolerant Distributed Software Families Development, in *OOPSLA and GPCE Workshop on Best Practices for Model Driven Software Development*, 2004.
- 34. P. H. da S. Brito, C. R. Rocha, F. C. Filho, E. Martins and C. M. F. Rubira, A method for modeling and testing exceptions in component-based software development, in *LADC*, 2005.
- 35. F. C. Filho, P. H. Brito and C. M. Rubira, Modeling and Analysis of Architectural Exceptions, in *Workshop on Rigorous Engineering of Fault Tolerant Systems Event Information, in conjunction with Formal Methods 2005. 18-22 July 2005, University of Newcastle upon Tyne, UK*, 2005.
- 36. R. de Lemos, Idealised Fault Tolerant Architectural Element, in *DSN 2006 Workshop on Architecting Dependable Systems (WADS06)*, 2006.
- 37. M. Shaw and P. Clements, A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems, in *COMPSAC97, 21st Int. Computer Software and Applications Conference*, 1997.
- 38. R. de Lemos, P. A. de C. Guerra and C. Rubira, A Fault-Tolerant Architectural Approach for Dependable Systems *IEEE Software, Special Issue on Software Architectures* March/April 2006.
- 39. V. Issarny and J. Banatre, Architecture-based Exception Handling, in *HICSS '01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 9*, (IEEE Computer Society, Washington, DC, USA, 2001).
- 40. Y. Feng, G. Huang, Y. Zhu and H. Mei, Exception Handling in Component Composition with the Support of Middleware, in *ACM Proc. Fifth International Workshop on Software Engineering and Middleware (SEM 2005)*, 2005.
- 41. A. Bondavalli and S. Chiaradonna and D. Cotroneo and L. Romano, Effective fault treatment for improving the dependability of COTS- and legacy-based applications *IEEE Transactions on Dependable and Secure Computing* **vol. 1, numb. 4**, 2004.
- 42. D. M. Beder, A. Romanovsky, B. Randell and C. M. Rubira, On Applying Coordinated Atomic Actions and Dependable Software Architectures in Developing Complex Systems, in *4th IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC'01)*, (Magdeburg, Germany, 2001).
- 43. A. F. Garcia and C. M. Rubira, *Advances in Exception Handling Techniques* (Springer-Verlag, LNCS-2022, April 2001), ch. An Architectural-based Reflective Approach to Incorporating Exception Handling into Dependable Software.
- 44. A. Capozucca, N. Guelfi, P. Pelliccione, A. Romanovsky and A. Zorzo, CAA-DRIP: a framework for implementing Coordinated Atomic Actions, in *The 17th IEEE International Symposium on Software Reliability Engineering (IS-SRE 2006)*, 7-10 November 2006 - Raleigh, North Carolina, USA.
- 45. P. Ammann, S. Jajodia and P. Liu, *A Fault Tolerance Approach to Sur-*

*vivability*, tech. rep., Center for Secure Information Systems, George Mason University (April 1999).

46. Object Management Group, OMG/Unified Modelling Language(UML) V2.0 (2004).
47. G. J. Pai and J. B. Dugan, Automatic Synthesis of Dynamic Fault Trees from UML System Models *13th International Symposium on Software Reliability Engineering (ISSRE'02)* (IEEE Computer Society, Los Alamitos, CA, USA, 2002).
48. A. Bondavalli, I. Majzik and I. Mura, Automatic dependability analysis for supporting design decisions in UML, in *Proc. of the Fourth IEEE International Symposium on High Assurance Systems Engineering*, eds. R. Paul and C. Meadows (IEEE, 1999).
49. A. Bondavalli, M. D. Cin, D. Latella, I. Majzik, A. Pataricza and G. Savoia, Dependability analysis in the early phases of UML-based system design. *Comput. Syst. Sci. Eng.* **vol. 16**, numb. **5**, 2001.
50. I. Majzik, A. Pataricza and A. Bondavalli, Stochastic dependability analysis of system architecture based on uml models, in *Architecting Dependable Systems, LNCS 2677*, eds. R. De Lemos, C. Gacek and A. Romanovsky Lecture Notes in Computer Science (Springer-Verlag, Berlin, Heidelberg, New York, 2003) pp. 219–244.
51. E. M. Clarke, O. Grumberg and D. A. Peled., *Model Checking* (The MIT Press, Massachusetts Institute of Technology, 2001).
52. C. Bernardeschi, A. Fantechi and S. Gnesi, Model checking fault tolerant systems *Software Testing Verification and Reliability* **vol. 12**, 2002.
53. T. Yokogawa, T. Tsuchiya and T. Kikuno, Automatic verification of fault tolerance using model checking, in *Proc. 2001 Pacific Rim International Symposium on Dependable Computing*, (IEEE Computer Society, Los Alamitos, CA, USA, 2001).
54. G. Bruns and I. Sutherland, Model checking and fault tolerance, in *AMAST '97: Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology*, (Springer-Verlag, London, UK, 1997).
55. H. R. Andersen, Partial model checking, in *LICS '95: Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*, (IEEE Computer Society, Washington, DC, USA, 1995).
56. S. Gnesi, G. Lenzini and F. Martinelli, Logical specification and analysis of fault tolerant systems through partial model checking, in *Proceedings of the International Workshop on Software Verification and Validation (SVV 2003), Mumbai, India*, eds. S. Etalle, S. Mukhopadhyay and A. Roychoudhury (Elsevier, Amsterdam, December 2003).
57. J. J. Klijach, B. T. Smith and A. S. Wojcik, Formal verification of fault tolerance using theorem-proving techniques *IEEE Trans. Comput.* **vol. 38**, numb. **3** (IEEE Computer Society, Washington, DC, USA, 1989).
58. L. Pike, J. Maddalon, P. Miner and A. Geser, Abstractions for fault-tolerant distributed system verification, in *Theorem Proving in Higher Order Logics (TPHOLs)*, eds. K. Slind, A. Bunker and G. Gopalakrishnan, Lecture Notes in Computer Science, Vol. 3223 (Springer, 2004).

59. B. Bonakdarpour and S. S. Kulkarni, Towards reusing formal proofs for verification of fault-tolerance, in *AFM (Automated Formal Methods) August 21, 2006, Seattle, USA*,
60. S. Owre, J. Rushby, N. Shankar and F. von Henke, Formal verification for fault-tolerant architectures: Prolegomena to the design of pvs *IEEE Transactions on Software Engineering* **vol. 21, numb. 2** (IEEE Computer Society, Los Alamitos, CA, USA, 1995).
61. D. Jackson, Alloy: a lightweight object modelling notation *ACM Trans. Softw. Eng. Methodol.* **vol. 11, numb. 2** (ACM Press, New York, NY, USA, 2002).
62. F. C. Filho, A. Romanovsky and C. M. F. Rubira, Verification of coordinated exception handling, in *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, (ACM Press, New York, NY, USA, 2006).
63. J. Xu, B. Randell, A. Romanovsky, C. M. F. Rubira, R. J. Stroud and Z. Wu, Fault tolerance in concurrent object-oriented software through coordinated error recovery, in *FTCS '95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, (IEEE Computer Society, Washington, DC, USA, 1995).
64. B. Randell, System Structure for Software Fault Tolerance *IEEE Transactions on Software Engineering. IEEE Press SE-1*, 1975.
65. A. Bertolino, Software testing research and practice, in *Abstract State Machines 2003. Advances in Theory and Practice: 10th International Workshop, ASM 2003, Taormina, Italy, March 3-7, 2003. Proceedings. Book Series Lecture Notes in Computer Science. Volume 2589/2003*, (Springer Berlin / Heidelberg, 2003).
66. M. R. Lyu, Z. Huang, S. K. S. Sze and X. Cai, An empirical study on testing and fault tolerance for software reliability engineering, in *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, (IEEE Computer Society, Washington, DC, USA, 2003).
67. A. Buccharone, H. Muccini and P. Pelliccione, Architecting fault-tolerant component-based systems: from requirements to testing, in *2nd VODCA Views On Designing Complex Architectures proceedings. To appear on ENTCS (Electronic Notes in Theoretical Computer Science)*, 2006.
68. J. Armstrong, R. Virding, C. Wikström and M. Williams, *Concurrent Programming in Erlang*, second edn. (Prentice-Hall, 1996).
69. D. Detlefs and M.P. Herlihy and J.M. Wing, Inheritance of synchronization and recovery properties in avalon/c++ *IEEE Computer* **vol. 21, numb. 12**, 1988.
70. A. Z. Spector, R. F. Pausch and G. Bruell, Camelot: A flexible distributed transaction processing system., in *COMPCON'88, Digest of Papers, Thirty-Third IEEE Computer Society International Conference, San Francisco, California, USA, February 29 - March 4, 1988*, (IEEE Computer Society, 1988).
71. B. Liskov, The argus language and system, in *Distributed Systems: Methods and Tools for Specification, An Advanced Course, April 3-12, 1984 and April 16-25, 1985 Munich*, (Springer-Verlag, London, UK, 1985).
72. B. Randell and J. Xu, Object-Oriented Software Fault Tolerance: Frame-

work, Reuse and Design Diversity *Predictably Dependable Computing System (PDCS 2) First Year Report*, 1993.

73. P. Rogers, Software Fault Tolerance, Reflection and the Ada Programming Language, PhD thesis, Department of Computer Science, University of YorkOctober 2003.
74. J. Kienzle and R. Guerraoui, AOP - Does It Make Sense? The Case of Concurrency and Failures, in *16th (ECOOP'2002)*, ed. B. Magnusson, (2374) (Malaga, Spain, 2002).
75. S. Soares, E. Laureano and P. Borba, Implementing distribution and persistence aspects with AspectJ, in *Proceedings of the 17th ACM Conference on Object- Oriented Programming, Systems, Languages, and Applications*, (ACM Press, 2002).
76. A. Rashid and R. Chitchyan, Persistence as an aspect, in *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development - AOSD'2003*, ed. M. Aksit (ACM Press, March 2003).
77. J. Kienzle, *Open Multithreaded Transactions — A Transaction Model for Concurrent Object-Oriented Programming* (Kluwer Academic Publishers, 2003).
78. J. Fabry and T. Cleenewerck, Aspect-oriented domain specific languages for advanced transaction management, in *International Conference on Enterprise Information Systems 2005 (ICEIS 2005) proceedings*, (Springer-Verlag, 2005).
79. R. Johnson, <http://st-www.cs.uiuc.edu/users/johnson/frameworks.html> (1997).
80. R. Martin and S. Totten, Introduction to Fault Tolerant CORBA (2003).
81. D. Gelernter, Generative communication in linda *ACM Trans. Program. Lang. Syst.* **7** (ACM Press, New York, NY, USA, 1985).
82. D. Bakken and R. Schlichting, Supporting fault-tolerant parallel programming in linda *IEEE Transactions on Parallel and Distributed Systems* **6**, 1995.
83. B. Arief and A. Iliasov and A. Romanovsky, On Using the CAMA Framework for Developing Open Mobile Fault Tolerant Agent Systems, in *SELMAS 2006 Workshop, as part of the 28th International Conference on Software Engineering*, (ACM Press, 2006).
84. Jini JavaSpaces, <http://www.sun.com/software/jini/specs/jini1.2html/jsspec.html>.
85. Jini, <http://www.sun.com/software/jini/>.
86. L. L. Pullum, *Software fault tolerance techniques and implementation* (Artech House, Inc., Norwood, MA, USA, 2001).
87. G. D. Parrington, S. K. Shrivastava, S. M. Wheater and M. C. Little, The Design and Implementation of Arjuna *Computing Systems* **vol. 8, numb. 2**, 1995.
88. J. Kienzle, Open Multithreaded Transactions: A Transaction Model for Concurrent Object-Oriented Programming, PhD thesis, Swiss Federal Institute of Technology Lausanne (Thesis no 2393)April 2001.
89. A. F. Zorzo and R. J. Stroud, A distributed object-oriented framework for

dependable multiparty interactions, in *OOPSLA '99*, (ACM Press, 1999).

- 90. A. Zorzo, Multiparty Interactions in Dependable Distributed Systems, PhD thesis, University of Newcastle upon Tyne, Newcastle upon Tyne, UK1999.
- 91. M. Evangelist and N. Francez and S. Katz, Multiparty interactions for inter-process communication and synchronization *IEEE Transactions on Software Engineering* **15**, 1989.
- 92. Y.-J. Joung and S. A. Smolka, A comprehensive study of the complexity of multiparty interaction *J. ACM* **43** (ACM Press, New York, NY, USA, 1996).
- 93. F. Cristian, Exception Handling *Dependability of Resilient Computers* (ed. T. Anderson) (Blackwell Scientific Publications, 1989).
- 94. J. Vachon, COALA : a design language for reliable distributed systems, PhD thesis, Swiss Federal Institute of Technology Lausanne (Thesis no 2302)2000.
- 95. B. Randell, A. Romanovsky, R. J. Stroud, J. Xu and A. F. Zorzo, *Coordinated Atomic Actions: from Concept to Implementation*, Tech. Rep. 595 (1997).
- 96. G. D. M. Serugendo, J. Fitzgerald, A. Romanovsky and N. Guelfi, Towards a Metadata-Based Architectural Model for Dynamically Resilient Systems, in *SAC'07, March 11-15, Seoul, Korea*, (ACM Press, 2007).